[56129050-3]

Application

For

United States Letters Patent

To all whom it may concern:

Be it known that I,

Bruce Hodge

have invented certain new and useful improvements in

OBJECT TYPE-DECLARATION PREFIX SYNTAX

of which the following is a full, clear and exact description:

Eunhee Park
Reg. No. 42,976
Baker & McKenzie
805 Third Avenue
New York, NY 10022

Express Mail Label No.: EK052645465US

OBJECT TYPE-DECLARATION PREFIX SYNTAX

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application claims benefit of the filing date of U.S. Patent Application No. 60/136,957 entitled DYNAMIC OBJECT SYNTHESIS WITH AUTOMATIC LATE BINDING, filed on June 1, 1999.

TECHNICAL FIELD OF THE INVENTION

The present invention relates generally to computer programming language and particularly, to system and method of declaring object type information in a programming language.

BACKGROUND OF THE INVENTION

Sub B3

Most of the current scripting languages on the market use Open DataBase Connectivity ("ODBC") to manipulate databases. ODBC allows a user to transfer data to and from databases using the Standard Query Language ("SQL"). Both ODBC and SQL are well known standards in programming industry. However, both languages are complex and require extensive learning and practice before they can be used. An average developer of software, e.g., World Wide Web ("web") developer typically does not have the programming expertise required to perform the complex functions associated with ODBC. Therefore, it is highly desirable to have a programming language that would significantly enhance the clarity and reduce the number of necessary lines of code to implement a standard query.

Express Mail Label No.: EK052645465US
Date of Deposit: May 31, 2000

Sub A' The presently prevalent scripting languages, PERL and Java are very powerful, but are also very complex to use and very resource-intensive to accomplish even the simplest of tasks. Therefore, it is highly desirable to have a programming language syntax that is simple to implement and yet efficiently cover most of what developers need to create high-end interactive applications.

SUMMARY OF THE INVENTION

The present invention is directed to a programming language syntax that embeds object type declaration in the object name. The objects are self-documenting because its object type is embedded in each object. By only examining the object names, therefore, methods associated with the object can be easily determined without resorting to the class definition for the object. Consequently, the language syntax in the present invention enables interpreters to process objects in an intelligent manner. For example, interpreters do not have to refer back to additional information or additional files having information about the objects because the object type is embedded in each object. Not accessing additional files or memory locations can greatly improve speed and efficiency of an interpreter.

With the present invention, programmers or writers of programming code need not explicitly declare variables. Consequently, the number of lines of code typically required in software code is reduced thereby reducing the time to develop and implement software code. Reduced lines of code also means less memory space required to store the code. The present invention, therefore, can operate with much less memory or disk space than is required by the existing conventional programming languages. Because the present invention need not have explicit declarations for each variable used, developing software code becomes faster and easier.

Sub a2 ~~Moreover, because the type-declarations are embedded with the object names, thereby rendering the objects self-contained, the code including type declaration of the present invention can be easily embedded and/or ported into a code of another language, for example, hypertext markup language ("HTML"). The present invention can also be easily integrated into an interactive development environment ("IDE") such as Visual Basic and Java Symantic Cafe.~~

Moreover, the present invention enables programmers and/or developer to easily identify and isolate errors by visual inspection, thereby enhancing greater ease in debugging codes.

In one embodiment, the object type-declaration prefix precedes the object name and explicitly declares the object type information. For example, a SQL object FirstName, may be implemented as SQL@FirstName, a URL object may be implemented as URL@FirstName, environment object from an environment table may be implemented as ENV@user.

Further features and advantages of the present invention as well as the structure and operation of various embodiments of the present invention are described in detail below with reference to the accompanying drawings. In the drawings, like reference numbers indicate identical or functionally similar elements.

BRIEF DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the present invention will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 shows object names having object type-declaration syntax of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention is directed to a programming language syntax that embeds object type declaration in the object name. The objects are self-documenting because its object type is embedded in each object. The present invention is related to a co-pending U.S. Patent Application Serial No. __ (Attorney Docket No. 56129050-4) entitled DYNAMIC OBJECT SYNTHESIS WITH AUTOMATIC LATE BINDING, filed on May 31, 2000, the disclosure of which is incorporated herein in its entirety by reference thereto.

In one embodiment, the object type-declaration prefix precedes the object name and explicitly declares the object type information. Figure 1 shows object names having object type-declaration syntax of the present invention. For example, a SQL object FirstName, may be implemented as SQL@FirstName by concatenating the object type SQL 102 with object name FirstName 106. A joint attribute or a separator may be an @ symbol 104. Similarly, a URL object may be implemented as URL@FirstName by concatenating the object type URL 108 with object name FirstName 112. A joint attribute or a separator may be an @ symbol 110. Environment object from an environment table may be implemented as ENV@user by concatenating the object type ENV 114 with object name user 118 and a joint attribute @ symbol 116.

In the above example, the objects are self-documenting because its object type precedes the name of the object. The following syntax creates a variable object called SQLcmd that has a select command, where the account number was sent from the previous page in a URL object called Acc and where the userId was captured from the environment table by a web server.

```
    var@SQLcmd = "select Id from accessLog where acc
="+URL@Acc+" and userId = "and
userId="'+ENV@REMOTE_USER+"'"
```

In the following syntax, using the connection object called myConnection, SQL statement stored in the object SQLcmd is prepared. The cursor returned from myConnection's prepare method is assigned to the cursor object named myCursor.

```
    CURSOR@myCursor =
CONN@myConnection.prepare(var@SQLcmd)
```

In the next statement, cursor object is used to fetch the Id in the table and print the information out. SQL@Id is the column name of the returned data.

```
    CURSOR@myCursor.fetch();
    Print("The user", ENV@REMOTE-USER, "Id from accessLog
where acc =", URL@Acc, "is", SQL@Id);
```

As described, the object type-declaration prefix syntax ("OTDPS") represents the type of object embodied within the object's name. In one embodiment of the present invention, a literal string embedded with the name of the object declares the data type. Using a literal string makes the present invention far more extensible than any prior art methods.

```
    Cursor@Clients = Conn@ClientList.prepare ("select
FirstName, LastName, Pone from ClientLIst");
     while (Cursor@Clients.fetch())
          {print (SQL@FirstName, SQL@LastName, SQL@Phone);}
```

In the example above, it can be appreciated that a developer is able to recognize specific methods applied to particular objects. For example, a cursor object may be created using a connection object's prepare method. Without the Conn@ prefix before the Clientlist, however, it would be difficult to know that ClinetList is a Connection object. Similarly, without the cursor@ OTDPS specified in the object name, it would be difficult to know to call the Clients fetch method to retrieve the selected data from the database. Further, the SQL@ OTDPS specified in front of the FirstName, LastName, and Phone objects represents that the data returned from the Clients fetch is SQL objects.

Examples of the language command set that includes the type declaration syntax of the present invention are described herein below in greater detail.

Sub B1

1. Syntax

<% DynaScript %>

These markers delineate a DynaScript code within an HTML document. This can occur more than once in a document, wherever dynamic content is needed.

/* multi-line comment */

All text between these markers are treated as a comment.

//<single line comment>

All text following this marker is treated as a comment up to the end of the line.

2. Literals

numbers	1, 2.3, .444444, 5.6789e+12, 0777, 0x12df
strings	'string"A"', "B string=john's name",
escape	\b, \f, \n, \r, \"
boolean	true, false
null	

3. Variables

Variables have this format:

```
<type>@<identifying name>
```

There are several basic types of variables used, and they are easily identified by their prefix (i.e., var@Cnt, URL@Name, SQL@Phone, ...).

The present invention includes three base classes or variables. Automatic classes are variables that are automatically created. They come into existence when first given a value, either through an assignment or function call. Created classes are variables created explicitly using the CreateObject(). Default classes are objects that are created by default whenever the script is run.

var@<variable identifying name> is a general purpose automatic variable. Assigning a value to a var@object creates a variable with a scope global to the entire execution. It may be destroyed when the execution ends. Examples of the var@ variables include: var@today="2000-06-01" which creates and/or sets a variable to today's date; var@year = var@today.substr(0,4) which creates and/or sets a variable to the year from a date; and var@Cnt=2 which creates and/or sets variable to value 1.

4. Expression statement

Typical expression statements in the present invention may include: !var@flag, var@Cnt!=1, exists(ENV@DBQUERY), and var@xcount=URL@URL.length().

5. Statements

In one embodiment, the present invention includes singular statements as well as compound statements. An example of a singular statement is var@Cnt=1. A compound statement is a set of statements that may be combined into a single statement by enclosing the statements in braces. An example include { var@Cnt++; print(var@Cnt) }.

6. Control logic

6.1 If...elseif...else

This clause executes selected statements when an expression evaluates to true. If no expression evaluates to true, a default set of statements under the else is executed. The elseif and else clauses are optional. Multiple elseif clauses may exist. For example:

```
if (expression) single-statement;
elseif (expression) single-statement;
elseif (expression) single-statement;
elseif (expression) single-statement.
```

The compound statements may also be used in the above example in lieu of any one of the single-statements.

6.2 while

The while clause in the present invention executes a set of statements if an expression evaluates to true. The set of statements are executed repeatedly until the expression evaluates to false. For example:

```
while (expression) single-statement; or
while (expression) {statement; statement;}.
```

6.3 do while

The do while clause in the present invention executes a set of statements at least once. If n expression evaluates to true, the set of statements are executed again. For example:

```
do statement while (expression); or
do {statement; statement;} while (expression).
```

6.4 for

The for clause in the present invention is shorthand for a typical while loop. It creates a loop that has one or more, preferably, three option expressions enclosed in parentheses and separated by semicolons. For example:

```
for (initialize; test; increment) single-statement; or
for (initialize; test; increment) {
    statement; statement; }.
```

6.5 labels

The labels in the present invention enables loop statements to be identified. Exit and Continue statements may target loops at higher nesting levels using labels. For example,

```
identifier: do / while / for clause
loop1: while (true).
```

6.6 exit <labelname>

The exit statement followed by a label in the present invention enables a control to exit a loop such as while, do-while, and for statements, and immediately to execute the next statement following the close of the loop clause. If no label follows the exit statement, then the current loop is exited. For example,

```
while (var@loop1<0)
{ var@loop1++;
  if (var@loop1 == 5) exit;
}
```

Another example may be,

```
loop1: while (var@loop1 < 5)
{ var@loop1++; var@loop2=0;
  loop2: while (var@loop2 < 5)
     { var@loop2++; var@loop3=0;
       loop3: while (var@loop3 < 5)
          { var@loop3++;
            if (var@loop3 == 2) exit loop1;
            var@Cnt++;
          }
     }
}
```

6.7 continue <labelname>

The continue statement in the present invention enables program control to exit a loop such as while, do-while, and for clauses, and immediately begin the next iteration of the same loop. Labels may be used to specify which loop when working with nested loops. When a continue statement is encountered, the execution of the subsequent code within a loop is skipped and program control is returned to the beginning of the loop to begin execution of next iteration. If no label follows the continue statement, then the current loop is used as a default loop. For example,

```
while (var@loop < 10)
{ var@loop1++;
  if (var@loop1 > 5) continue;
var@Cnt++;
}

loop1: while (var@loop1 < 5)
     { var@loop1++; var@loop2=0;
       loop2: while (var@loop2 < 5)
```

```
          { var@loop2++; var@loop3=0;
          loop3: while (var@loop3 < 5)
             { var@loop3++;
               if (var@loop3 == 2) continue loop2;
               var@Cnt++;
             }
          }
        }
```

6.8 end

The end statement in the present invention immediately terminates a program.

7 Operators

Table 1 includes assignment operators used in one embodiment of the present invention.

Table 1

Operators			

=	Assignment	X=0
		Y=X=2
			Assigns a value to a variable

++	Increment	x=2; ++x (returns 3, x==3)	Decrements operand by one and returns new value (++i) or the old value (i++)

-	Decrement	x=2; --x (returns 1, x==1)
		x=2; x-- (returns 2, x==1)
			Decrements operand by one and returns new value (--i) or the old value (i--)

+	Addition String Concatenate	x=2;y=3;x+y (returns 5) "str1"+"str2" (returns "strstr2")	Adds two operands and returns sum (i+y) Concatenates two strings ("str1"+"str2")

_	Subtraction
	Unary Negation
		x=3;y=2;x-y (returns 1)
		x=3;-x(returns -3)
			Substracts two operands and returns difference (i-y)
			Converts positive operands to negative (-i)

*	Multiplication	5*2 (returns 10)	Multiplies two operands and returns result (i*y)
/	Division	15/5 (returns 3)	Divides two operands and returns result (i/y)

%	Modulo	15.3 % 5 (returns 0.3)	Divides two operands and returns remainder (i%y)
&	Bitwise AND	7 & 2 (returns 2)	Bits different are set to 0
|	Bitwise OR	4 | 1 (returns 5)	Bits different are set to 1
^	Bitswise XOR	3 ^ 2 (returns 1)	Bits different are set to 1, all other set to 0
~	Bitwise NOT	~ 5 (returns -4)	Bits are reversed from their original value (ones compliment)

&&	Logical AND	(1==1) && (2==2) (returns true) (1==1) && (2==3) (returns false)	Returns a Boolean value and the operand expressions are evaluated as Boolean expressions. True only if both operands evaluate to True, otherwise false
||	Logical OR	(1==1) || (2==3) (returns true) (1==2) || (2==3) (returns false)	Returns a Boolean value and the operand expressions are evaluated as Boolean expressions. True only if both operands evaluate to True, otherwise false
!	Logical NOT	!(1==1) (returns false)	Inverts the Boolean value of the expression evaluated

==	Equality	5==5(returns true) 5==4(returns false)	True only if operands are the same value
!=	Inequality	5!=5(returns false) 5!=4(returns true)	True only if operands are NOT the same value
<	Less than	5<5 (returns false) "4" < "5" (returns true)	True if first operand is less than the second operand
<=	Less than or equal	5.1 <= 5 (returns false) 5 <=5 (returns true) "4" <= "5" (returns true)	True if first operand is less than or equal to the second operand
>	Greater than	5 > 5 (returns false) "5" > "4" (returns true)	True if first operand is greater than the second operand
>=	Greater than or equal	5.1 >= 5 (returns true) 5 >= 5 (returns false) 5" >= "4" (returns true)	True if first operand is greater than or equal to the second operand

[]	Array index	URL@Name[var@Cnt].value	Allows access to individual elements of an array. Zero is the first element.

7.1 Assignment shorthand

In one embodiment, the present invention also may include shorthand assignment operator notations. For example, combinations of the assignment operator and other operators may be written in shorthand notation as shown in Table 2. Table 2 lists the operators or the present invention in one embodiment.

Table 2

x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y
x%=y	x=x%y
x&=y	x=x&y
x^=y	x=x^y
x|=y	x=x|y
<<=	x=x<<y
>>=	x=x>>y

8 Functions

Functions include calls that do not use "variable type declaration prefix" notation, e.g., VAR@string.length().

8.1 nbsb (<mode>)

<mode>   1 = turn on, 0 = turn off

<returns> none

nbsb in one embodiment of the present invention represents a none breaking space mode. If this mode turned on, NULL characters, e.g., returned from a database query results, may be automatically replaced by "&nbsp". This automatic replacement method is useful when populating cells in a web page table.

8.2 print (<format>, <variable list>)

<format>   standard "C" printf format string

<variable list>   set of optional variables separated by commas, for example

<returns>   formatted string

The print function in the present invention sends a formatted string to standard output. The string may be created using the same format notation used for the printf in standard ANSI C. Examples include: print (Count Integer: %d\n", SQL@cnt); print ("Float: %6.3f\n" var@floatVal);

8.3 sprint (<format>, <variable list>)

<format>   standard C printf format string

<variable list>   set of optional variable, may be separated by commas

<returns>   formatted string

The sprint of the present invention returns a formatted string, which may be created using the same format notation used for the sprint function in standard ANSI C. Example include: var@Str1 = sprint ("count

Integer: %d\n", SQL@cnt); var@Str2 = sprint ("Float: %6.3f\n", var@floatVal).

8.4 exists (<object>)

<object> any object

<returns> true if the object exists, otherwise returns false

The exists function determines whether an object has been created. Example of a code using the exists statement is: if (exists (CURSOR@CharityDeductions)) ...

8.5 errorMode (< mode >)

<mode> 1 = turn on, 0 = turn off

<returns> none

The errorMode function sets ODBC to report errors verbosely and end program execution upon an occurrence of an error. An example usage of the errorMode is: errorMode(1).

8.6 alert (< text message >)

<text message>	a printable text string
<returns>	none

sub b2 The alert function causes a message to be displayed within a standard JavaScript dialog box. The dialog remains until the user responds by clicking on the "OK" button. An example usage of the alert function is: alert ("An error has ocurred\n" + "Please click on OK below to continue");

9 Environment object

The environment object type is generated by default at the start of a program execution. An ENV@ object may be created for each environment variable currently defined in the OS environment, e.g., DOS, Window, UNIX. A typical format is ENV@<name of environment variable> An example of a code using the environment object is: if (exist (ENV@REMOTE_USER)) var@str = ENV@REMOTE_USER.

9.1 Standard environment variables

Table 3 shows a list of standard environment variables supported in the present invention. Column 1 lists an operating system environment variable name, column 2 describes the variable, and column 3 list the variable name as used in the present invention.

Table 3

Operating system environment variable name	Description	Script variable name
GATEWAY_INTERFACE	The revision of the CGI that the server uses	ENV@GATEWAY_INTERFACE
SERVER_NAME	The server's hostname or IP address	ENV@SERVER_NAME
SERVER_SOFTWARE	The name and version of the server software that is answering the client request	ENV@SERVER_SOFTWARE
SERVER_PROTOCOL	The name and revision of the information protocol the request came with	ENV@SERVER_PROTOCOL

SERVER_PORT	The port number of the host on which the server is running	ENV@SERVER_PORT
REQUEST_METHOD	The method with which the information request was issued	ENV@REQUEST_METHOD
PATH_INFO	Extra path information passed to a CGI program	ENV@PATH_INFO
PATH_TRANSLATED	The translated version of the path given by the variable PATH_INFO	ENV@PATH_TRANSLATED
SCRIPT_NAME	The virtual path (e.g. /cgi-bin/program.pl) of the script being executed	ENV@SCRIPT_NAME
DOCUMENT_ROOT	The directory from which web documents are served	ENV@DOCUMENT_ROOT
QUERY_STRING	The query information passed to a program. It is appended to the URL with a "?"	ENV@QUERY_STRING

REMOTE_HOST	The remote hostname of the user making the request	ENV@REMOTE_HOST
REMOTE_ADDR	The remote IP address of the user making the request	ENV@REMOTE_ADDR
AUTH_TYPE	The authenication method used to validate a user	ENV@AUTH_TYPE
REMOTE_USER	The authentication name of the user	ENV@REMOTE_USER
REMOTE_IDENT	The user making the request. This variable is only be set if NCSA Identity Check flag is enabled, and the client machine support the RFC 931 identification scheme (ident daemon)	ENV@REMOTE_IDENT
CONTENT_TYPE	The MIME type of the query data, such as "text/html"	ENV@CONTENT_TYPE

CONTENT_LENGTH	The length of the data (in bytes or the number of characters) passed to the CGI program through standard input	ENV@CONTENT_LENGTH
HTTP_FROM	The email address of the user making the request. Most browsers do not support this variable	ENV@HTTP_FROM
HTTP_ACCEPT	A list of the MIME types that the client can accept	ENV@HTTP_ACCEPT
HTTP_USER_AGENT	The browser the client is using to issue the request	ENV@HTTP_USER_AGENT
HTTP_REFERE	The URL of the document that the client points to before accessing the CGI program	ENV@HTTP_REFERER
other environment variable may also be included		

10 URL object

Uniform resource locator ("URL") objects are created from a calling page. For example, CreateObject("URL") may be created to synthesize URL variables. URL objects typically have string type data structure. An example usage of a URL object in the present invention is URL@<URL identifier>. An example of a code using the URL object includes URL@myURL = CreateObject('URL');

10.1 Default URL object

URL@URL

URL@URL is a URL@ object representing a reserved symbol. Properties of URL@URL are described below.

URL@URL.length returns the number of key-value pairs in the URL@URL object. URL@URL.query returns the un-decoded URL query string. URL@URLname [<index>] returns the name of key-value pair name found at index location. <index> may be an integer index value with first element = 0. The URLname method returns a string value. Examples of use include var@KeyName1=URL@URL[0].name; and var@KeyName2=URL@URL[1].name.

URL@URLvalue [<index>] returns the value of key-value pair found at index location. <index> may be an integer index value with first element = 0. The URLname method returns a string or numeric value. Examples of use include var@KeyValue1=URL@URL[0].value; and var@KeyValue2=URL@URL[1].value.

URL@URL.encode (<un-encode string>) method returns a URL encoded string. <un-encode string> is a string. An example of use include var@myEncodedString = URL@URL.encode(var@aString).

URL@URL.decode (<URL string>) method returns a decoded URL string. <URL string> is a url string which needs to be decoded. An example of the method call includes: var@decodedQuery = URL@URL.decode(URL@URL.query).

URL@URL.encrypt (<string to encrypt>) method returns a string which has been encrypted. <string to encrypt> is a literal string or string variable. This method is useful in enabling secure communication of strings. Examples of use include: var@mycypherstring = URL@URL.encrypt(var@myPlainString); and var@u = "http://www.dynascipt.com?Name="+URL@URL.encrypt(var@Val).

URL@URL.decrypt (<string to decrypt>) method returns a string translated back into readable characters. The string may have been originally generated by the encrypt method. <string to decrypt> is a literal string or string variable in an encrypted format. An example of use of the method include: var@myPlainstring=URL@URL.decrypt(var@myCypherString).

URL@URLvalidate ([<URL>]) method returns true if the supplied URL is a properly formed URL. <URL> is a URL string. An example of use of the method include: if (URL@URL.validate (var@UsersURL)).

10.2 value

<parameters> none
<returns> string

The value function returns the value of the corresponding name. Multiple values may be passed for the same name, therefore, an index may be used to access each value. Examples of statements using the value function include: var@KeyValue = URL@KeyName.value; var@KeyValue1 = URL@KeyName[0].value; var@Keyvalue2 = URL@KeyName[1].value.

10.3 length

<parameter> none
<returns> integer value

The length function returns the number of values associated with a key name. Examples include: var@elements = URL@KeyName.length.

11 HTML object

11.1 Default HTML object

The HTML@ object may be created by default when a program is executed. In one embodiment of the present invention, it is a reserved symbol.

The HTML@ object include header, redirect and encode functions. For example, HTML@HTML.header [<header spec>] method outputs an HTML header needed at the beginning of all HTML pages. <header spec> specifies an optional, specific type of header whose default is a standard HTML header. The method returns nothing. This method is automatically evoked if the script starts outputting without sending an HTML header. Example usage includes: HTML@HTML.header () and HTML@HTML.header('text/plain').

The HTML@HTML.redirect (<URL>) method directs the users browser to retrieve an HTML page from another URL. <URL> represents a fully declared URL. The method returns nothing. Example usage includes: HTML@HTML.redirect ('http://www.dynascript.com').

The HTML@HTMLencode (<string to encode>) method outputs a HTML encode string. <string to encode> is a literal string or string variable to be HTML encdoed. The method returns nothing. Example usage includes: var@myString = HTML@HTML.encode (var@HtmlString).

12 Date object

The date object, DATE@<date identifier>, includes datetime information and methods. Example usage includes Date@lastUpdate = CreateObject('Datetime'). Its internal format may be represented as follows: yyyy-MM-dd hh:mm:ss.sss in US date style and 24 hour clock; 2000-04-15 23:59:59.012 which represents April 15th at the last second. The methods listed in Table 4 may be applied to the Date@object.

Table 4

getYear()	setYear(1999)	Four digit year (1999)
getMonth	setMonth(11)	Month(01-12)
getDate()	setDate('05/21/1999')	Date(<US style Date>)
getHours()	setHours()	Hours (24 Hour Style)
getMinutes()	setMinutes()	Minutes (00-60)
getSeconds	setSeconds	Seconds(00.000-60.000)
getTimeZone()	setTimeZone()	TimeZone Hours Offset from GMT (0,+/-01.0 to 12.0) (defaults to system TZ)
getDay()		Day of the week (0=Sun, 6=Sat)

12.1 DateTime set & get method summary

The format(<pattern>) method converts a DateTime object into a formatted string. <pattern> is a date format string. The method returns character string version of date. The returned string may be assigned to any string variable. Examples inlcude: var@Date = DATE@myDate.format("MM/dd/yy") where "MM/dd/yy" represent date having a form, 05/10/99, for example. <pattern> also may include "yyyy-MM-dd HH:mm:ss.sss" representing 1999-05-10 13:50:43.567 format, "hh:mm" representing 01:50 PM, "dd-MON-yy" representing 10-MAY-99, and "DAY MON dd,yyyy" representing Mon MAY 10, 1999. Table 5 includes a list of exemplary date formats in the present invention.

Table 5

Type	Pattern	Example
Year	yyyy	1999,2000,2001,...
Month	MM MON	01...12 'Jan'...'Dec'
Date	MM/dd/yyyy	setDate(01-12/01-31/1999-)
Day	DAY day	day('Sun','Mon','Tue',...)
Hour	hh HH	12 hour(01-12AM/PM)
Minute	mm	minutes (00-59)
seconds	ss	2 digit seconds (00-59)
Milliseconds	.sss	3 digit milliseconds (000-999)

13 String object

The string object in the present invention applies to character strings. The functions for operating on character strings are described below.

13.1 substr (<start position>, [<length>])

The substr method extracts a substring from a string. <start position> may an integer. For example, 0 represents a first character, and -1 represents a first character from the end of the string. <length> is a number of characters to extract. The default is the to the end of the string. Example usage includes: var@string.substr(0,4) and var@string.substr(0,var@cnt).

13.2 indexOf (<substring>,[<start>])

This method returns the starting location of a substring within a string. <substring> may be a single character, string, or string variable. <start> in an integer value indicating the starting point. For example, 0 represents first character, and -1 represents last character. The method returns an integer value of starting index position of substring in string. For example, 0 represent first position, and -1 represents string not found. Example usage includes: var@index = var@string.indexOf ('test').

13.3 trim (<string>)

This method removes leading and training spaces. It is typically intended for use with SQL queries, which may have additional spaces returned in the results. <string> is a string. The method returns a string. Example usage includes var@trimmed = trim (SQL@name).

13.4 length

This method returns the character count of the string object. The returned value is an integer length.

14 ODBC database object and methods

14.1 Single connection

Sub A4

~~when only a single connection is necessary, a simple syntax using the default connection and cursor object may be used. In this case, the syntax appears more like traditional C-like function calls. Figure ? illustrates an example. Another feature of the single connection approach is that SQL statements may be inserted into the script directly and need not be enclosed within a method call, e.g., prepare9). The bindparam, execute, and fetch methods of the default cursor may be invoked without an object declaration.~~

SuB a5

14.2 Multiple connection

Multiple connections may be created and managed simultaneously. Each connection may also have multiple result sets returned, which may be managed with cursors. Figure ? illustrates an example of code using the multiple connection.

14.3 SQL object

These variables include data that has been returned from a database. These objects have the format, SQL@<SQL name>.

14.4 Connection object

This variable represents a connection to a SQL server. Multiple database connections are allowed to be opened and queried simultaneously. The connection object has a format CONN@<connection identifier>. Examples of code using the connection object include: CONN@Greenfield = CreateObject ('Connection'); CONN@Greenfield.connect(" DSN=Greenfield; UID=webuser; PWD=hodgepodge"); CURSOR@MyCur = CONN@Greenfield.prepare ("select*from clientList").

14.4.1 Default connection object

The default connection object CONN@CONN may be created by default whenever a script is executed. This is the connection used when the single connection scripting style is used. When connection methods are called without a specific reference to a connection object, this default object is implied. Any CONN@ method may be invoked simply by calling the method as traditional function. For example, the object.method is not necessary and the call may be expressed as connect ("DSN=Greenfield;UID+webuser;PWD=hodgepodge") in place of

CONN@CONN.connect ("DSN=Greenfield;UID+webuser;PWD=hodgepodge").

14.4.2 connect (<database access string>)

<database access string> may include a series of semicolon (;) separated key-value pairs which define various connection parameters. The order of the key-value pairs is not significant. Key-value pairs may include DSN which is ODBC data source name, UID which is a user login, and PWD which is a user password. <alternate access string> represents string that does not explicitly declare DSN, UID, and password. When a UID and PWD are not required, this string may include only the data source name in quotes, e.g., "Greenfield". The connect method returns a connected property. Examples include CONN@Greenfield.connect("DSN=Greenfield; UID=webuser; PWD=hodgepodge"); CONN@Greenfield.connect("Greenfield"); and connect("DSN=Greenfield;UID=webuser;PWD=hodgepodge").

14.4.3 connected

The connected method returns true if the connection is connected, and false if it is not. The return value is typically a boolean true or false. Example includes: if (!CONN@Greenfield.connected).

14.4.4 disconnect ()

The disconnect method disconnects a connection. Connections are automatically disconnected upon exit. An example of the method call includes: CONN@Greenfield.disconnect ().

14.4.5 prepare (<SQL statement>)

The prepare method creates a cursor that can execute the SQL statement. <SQL statement> may be a quoted string which includes a single SQL statement. The method returns a Cursor object. An example of use include CURSOR@TaxCnt = CONN@Greenfield.prepare ("select count(*) from tax_info").

14.4.6 do <SQL statement>)

The do method performs a prepare and execute in one command. SQL statements that perform data inserts, updates or deletions use this command. <SQL statement> may include a quoted string which has a single SQL statement. This method returns a Cursor object. An example of use includes: CONN@Greenfield.do ("update tax_id set taxNo = 1234 where SSN = '123-45-6789'").

14.4.7 commit()

The commit method records changes to the database made in previous SQL statements since the last commit() or since the opening of the connection. Following this call, changes up to this point are recorded. The changes may not be discarded using rollback() method. If a connection has multiple cursors defined, changes made by all of those associated cursors are recorded. An example of use include: CONN@Greenfield.commit().

14.4.8 rollback()

The rollback method discards any changes to the database made in the previous SQL statements since ethe last commit() or since the opening of the connection. If a connection has multiple cursors defined, changes made by all of those associated cursors are discarded. An example rollback () method call includes: CONN@Greenfield.rollback().

14.4.9 setattr(<connection_attribute>,[<optional parameter>,..])

The setattr method is used to set the attributes of a connection. Parameters may be separated by comas. <connection_attribute> may be a standard ODBC attribute. <optional parameter> may include one or more optional parameters which are specific to the <connection_attribute>. An example of setattr method call includes: CONN@Greenfield.setattr("SQL_attribute_autocommit_on").

14.4.10 Connection attributes

The connection attributes include SQL_attribute_autocommit_on(default), SQL_attribute_autocommit_off, SQL_attribute_connection_timeout, SQL_attribute_login_timeout, SQL_attibute_trace. These attributes are typically defined by the ODBC standard.

14.5 Cursors

The Cursor object provides access to the results returned by an SQL query statement or function. Multiple cursor objects allow multiple results sets to be examined row by row simultaneously. This object has the following format: CURSOR@<cursor identifier>. Examples of use include: CURSOR@TaxId = CONN@Greenfield.prepare ("select tax_id from taxInfor"), and CURSOR@TaxId.execute().

14.5.1 Default connection object

CURSOR@CURSOR

This cursor object is created by default whenever a script is executed. This cursor may be used when retrieving results using the single connection scripting style. When cursor methods are called without a specific reference to a cursor object, this default object is implied. Examples of the method call include: select tax_Desc from tax_info where tax_id = 1234567890; execute (); fetch (tax_Desc). Any CURSOR@ methods may be invoked simply by calling the method as a graditional function; the object.method notation is not necessary. For example, fetch (SQL@name) may be used in place of CURSOR@CURSOR.fetch (SQL@name).

14.5.2 bindParam (<direction>,<script var>,<SQL param type>)

The bindParam method allows the cursor object to bind any variable to a parameter within the SQL statement. It generally is used to bind the arguments of stored procedure calls or select statements. The parameter is inserted in place of a "?" in the SQL statement. In one embodiment, a separate bindParam() call is needed for each "?" and they are associated in the order, i.e., first ? associated with the first bindParam() call. <direction> specifies the direction of data flow for the parameter. SQL_param_input is a variable used to pass data into the database. SQL_param_output is a varialbe used to pass data back from the database. SQL_param_input_output is a variable used to pass data to and from the database. <script var> includes a variable, e.g., URL@TaxId, var@myVar. <SQL param type> declares the SQL parameter type. An example of the bindParam call is: Cursor@TaxId = CONN@Greenfield.prepare ("select tax_id from taxInfo where name_last = ? and name_first = ?"); Cursor@TaxId.bindParam (SQL_param_input, URL@LastNm, SQL_varchar); Cursor@TaxId.bindParam (SQL_param_input, URL@FirstNm, SQL_varchar). Table 6 includes standard SQL parameter types.

Table 6

SQL_binginter	SQL_float	SQL_real
SQL_binary	SQL_int	SQL_smallint

SQL_bit	SQL_interval	SQL_timestamp
SQL_datetime	SQL_longbinary	SQL_tinyint
SQL_decimal	SQL_longvarchar	SQL_varbinary
SQL_double	SQL_numeric	SQL_varchar

14.5.3 bindCol ([<col number>], <script var>,)

The bindCol method binds a script variable with a column returned in a result set. Columns are specifed by a number related to the order in which column values are returned. If the column number is excluded, each consecutive bindCol() is associated with the corresponding column of the result set. <column number> is an optional column number and <script var> is a script variable, e.g., URL@TaxId, to which to bind. Example of use include Cursor@My.bindCol (1, var@lastUpdDate); and Cursor @My.bindCol (2, var@acctId).

14.5.4 setPos (<number>)

The setPos method sets the row position of a cursor pointing to a result set. This method may only be used if the SQL_attribute_scrollable attribute of the cursor has been turned on. The position within the result set is an absolute from either the first or last row. A positive number moves the cursor position forward and a negative number moves it backwards. An example of use include: CURSOR@CharityDeductions.setpos(2) which moves cursor forward by 2 rows.

14.5.5 execute ()

The execute method causes any previous SQL statements to be executed by the database server. An example of use include select tax_desc from tax_info where tax_id = 123456789; CURSOR@CURSOR.execute().

14.5.6 fetch (<SQL vars>)

Each time fetch method is called, it returns values for a single row of a result set. The SQL@ variables are assigned values in the order corresponding to the result columns, i.e., SQL@arg1 = colval1, SQL@arg2 = colval2, etc. If no variables are supplied variables are synthesized using the result set column names as the variables names. Examples of use include: CURSOR@TaxId = CONN@Greenfield.prepare ("select tax_id from taxInfo); while (CURSOR@TaxId.fetch()) { var@cnt ++; print("'%d, %s\n", var@cnt, SQL@taxId.substr(0,10)); }.

14.5.7 free()

The free method releases the results data managed by the cursor object. A call to free occurs automatically when another prepare call is made. An example of a call to the method includes: CURSOR@CharityDeductions.free().

14.5.8 setattr(<connection_attribut>,<optional parameter>, ...)

The seattr method is used to set the attributes of a cursor. Parameters may be separated by a comma. <connection_attribute> is a standard ODBC attribute, and <optional parameter> may include one or more optional parameters which are specific to the <connection_attribute>.

14.5.9 Cursor attribute

Table 7 includes a list of cursor attributes. These attributes are defined by the ODBC Standard 3.0.

Table 7

SQL_attribute_cursor_type	SQL_attribute_fetch_next

SQL_attribute_nonscrollable(default)	SQL_attribute_fetch_first
SQL_attribute_scrollable	SQL_attribute_fetch_last
SQL_attribute_forward_only (default)	SQL_attribute_fetch_prior
SQL_attribute_cursor_static	SQL_attribute_fetch_absolute
SQL_attribute_cursor_dynamic	SQL_attribute_fetch_relative

15 DB query language

The present invention is enabled to support a subset of the standard SQL query language. These statements are those that are likely to need to access a database from within a script of the present invention.

15.1 Standard SQL syntax allowed

Table 8 includes the SQL syntax supported in the present invention.

Table 8

Create a table	create table employee_list (id_number int(10) not null, lastname varchar(40) not null, firstname varchar(40) not null, phonenumber char(12) null)
Delete a table	drop table employee_list

Insert rows	insert into employee_list (id_number, lastname, firstname) values (121212, "Abelson:, "Alice") insert into employee_list (id_number, lastname, firstname) values (232323, "Benson:, "Bart") insert into employee_list (id_number, lastname, firstname) values (343434, "Clarkson:, "Clyde")
Delete rows	delete from employee_list where id_number = 343434
Update rows	update employee_list set phonenumber = "123-456-7890" where lastnmae = "Ableson", and firstname = "Alice"
Select from database	select * from employee_list order by lastname, firstname
Execute a stored procedure	exec GetAreaCode, 123 or BEGIN GetAreaCode (123) END;

16 Debugging

In the present invention, setting the debug mode at the start of a script allows the developer to output information that is not normally available. Debugging output is written to a separate file, e.g., debug.txt, in the same directory as the scripts executable so that it does not affect the normal output of the script.

DEBUG@DEBUG

DEBUG@DEBUG object is created by default at the start of a script in the present invention. The mode method is used to enable and disable the scripts debugging features.

The debug method has the following syntax: DEBUG@DEBUG.mode (<hexadecimal bit map>) where <hexadecimal bit map> is hexadecimal value (0xhhhhhhhh) where h is a hexadecimal digit (0-9, a-f). The hexadecimal values shown in Table 9 below may be combined or added into a single value.

Table 9

DEBUG_IO	0x00000001	Enable input and output of debugging information to debug.txt
not used	0x00000002	
not used	0x00000004	
not used	0x00000008	
DEBUG_ODBC	0x00000010	Set to dump ODBC commands
DEBUG_FETCH	0x00000020	Set to dump fetch variables
DEBUG_CALL	0x00000040	Set to dump procedure call variables
DEBUG_ASSIGNMENT	0x00000080	Set to dump all variable assignments
not used	0x00000100	
not used	0x00000200	
not used	0x00000400	
not used	0x00000800	
not used	0x00001000	
not used	0x00002000	
not used	0x00004000	
not used	0x00008000	
DEBUG_OUTPUT	0x00010000	Set to output debug data to client

DEBUG_FILE-APPEND	0x00020000	Set to output debug data to file in append mode
DEBUG_FILE_NEW	0x00040000	Set to output data to file in overwrite mode
DEBUG_PERSIST	0x00080000	Set to carry debug on to all other pages
not used	0x00100000	
not used	0x00200000	
not used	0x00400000	
not used	0x00800000	
not used	0x01000000	
not used	0x02000000	
not used	0x04000000	
not used	0x08000000	
not used	0x10000000	
DEBUG_AND_MASK	0x20000000	Set to AND out all lower non-set bits
DEBUG_OR_MASK	0x40000000	Set to OR in all lower bits
not used	0x80000000	

While the invention has been particularly shown and described with respect to a preferred embodiment thereof, it will be understood by those skilled in the art that the foregoing and other changes in form and details may be made therein without departing from the spirit and scope of the invention.